UCID- 20526

TEXT COMPRESSION USING WORD TOKENIZATION

Gary Long
Ira Morrison
David Barnett

September 11, 1985

Lawrence Livermore National Laboratory

Work performed under the auspices of the U.S. Department of Energy by the Lawrence Livermore Laboratory under Contract W-7405-Eng-48.

TEXT COMPRESSION USING WORD TOKENIZATION

Gary Long
Ira Morrison
David Barnett

Computations Department
Lawrence Livermore National Laboratory
University of California
Livermore, California

August 30, 1985

ABSTRACT

This document describes a text compression scheme, its associated algorithms, and their implementation in the 'C' programming language. The algorithms described only work for textual data. The current code and associated data dictionary are slanted toward general English language text such as technical reports and/or newspaper articles. With minor modifications to the code and by using special purpose dictionaries, the same algorithms could be used to compress special purpose text files such as compiler source listings. We achieve compression ratios of better than fifty percent and the routines are very fast. Typical applications produce significant reduction in data storage space and effectively double the transmission rate of documents between computers.

Text Compression Using Word Tokenization

1. INTRODUCTION

The rapid proliferation of large textual databases and the concomitant need to rapidly transmit textual data between computers prompted a reevaluation of existing data compression schemes. The compression schemes commonly used in the past have been based largely on information theoretic concepts such as minimizing redundancy, with little dependence on the specific content of the data being compressed. They tended to be small in size but difficult to efficiently implement in software. Since "fast" computer memory of the type necessary to implement dictionary encoding schemes continues to become less expensive, it seemed reasonable to develop a scheme based on the statistics of word (type) usage in the English language. The method reported on in this paper was designed by Gary Long and Ira Morrison and implemented by David Barnett. It is based on a word tokenization scheme. It uses modest amounts of memory (53K bytes of data and 14K bytes of code on an IBM PC, and 55K bytes of data and 28K bytes of code on a MicroVAX I) and the compression ratio is greater then 50% for English language text. For example, this paper requires 39K bytes of storage without compression. In compressed form, it only requires 18K bytes. The 50% compression ratio provides an effective doubling of the baud rate when used in data communications applications.

The code itself is written in 'C', and excerpts from the actual code are included.

2. ALGORITHM OVERVIEW

The algorithms incorporate two different methods to compact text. One is encoding of English words, that is, replacing each word with a number, where the number is an index to that word in a dictionary. The other is based on assumptions about punctuation and white space (e.g. tabs, carriage returns, spaces, etc.) which makes it possible to eliminate most spaces from a document being compressed. These two methods are outlined below.

2.1. Tokenization

The current implementation of this compression method uses a dictionary of the eight thousand (more precisely 8K) most common words in the English language. In addition, there is an auxiliary dictionary which contains the thirty-two most common words. The words in the auxiliary dictionary are represented by one byte (8 bits). The other 8K words are compressed into a two byte representation. (For information on the restriction of the number of available bits, see Appendix A.)

The algorithm not only uses literal word lookup but also applies tests for prefixes, suffixes, and accompanying root modification in order to compact word variations not contained literally in the dictionary. If a word variant is found in this manner, the prefix and/or suffix is encoded in one or more extra bytes which follow the encoded dictionary index of the root word. If a word or variant is not found in the dictionary, it is not encoded but is placed literally (i.e. character by character) in the compressed document.

Finally, the program also checks the case of the characters making up a word, and can encode that information. The recognized cases are all capitals, all lower case, and first letter capitalized. If a word is capitalized in any other way it can not be compressed. Nothing is ever compressed if it can not be identically decompressed later.

2.2. Punctuation and White Space

The algorithm also attempts to compact information concerning punctuation. It does this by making assumptions about each punctuation mark. Each punctuation mark is specifically tagged as either having a space or none before and/or after it. If a particular punctuation mark is tagged as being preceded and/or followed by a space and it actually is, then that space need not be included in a compressed document. However, if it is not accompanied by an expected space, a special character must be included in the compacted text to indicate the exception. Hence, a punctuation mark must be preceded or followed by a space more than half the time in order for effective compression to occur. If a space is not assumed before or after a punctuation mark and there is one, then it is just carried literally. For example, the percent sign (%) is tagged as having no space before it and one after it, so the pattern "a% b" would be compressed to "a%b" and "a % b" to "a %b".

Additionally, it is assumed that every word is followed by a space. Thus, spaces between words are not included in a compressed document (unless one falls between two words which could not be tokenized). The exception to this feature occurs when a word is followed by a punctuation mark not expected to be preceded by a space. Punctuation assumptions always have precedence over the "space after" assumption. Exceptions are noted in the compression.

One last compression effort is made by assuming that some punctuation marks not only are followed by a space but also end a sentence. This means that if the word following them begins with a capital letter, it is unnecessary to include this capitalization attribute in the compressed file.

2.3. Example

The following illustrates the gist of this scheme. Figure 1 contains a sentence, and below the sentence an indication of how it is compressed. The number of characters with a space underneath them is the number of saved characters, since each space represents a byte not required in the compressed sentence. In Figure 1 assume that an (*) represents a word encoded in one byte. A (##) represents a word encoded into two bytes. An (@) indicates the presence of an extra byte containing affix or capitalization information. A (-) indicates an uncompressed character. The original sentence containing 53 characters is compressed into one containing only 26 characters, yielding 51% compression. In reality, the common words "simply" and "amazing" could be in the dictionary literally, and so not require an extra byte to indicate the suffix. Common names like "Smith" could also be included in the dictionary. If "This" were preceded by an end of sentence marker rather than being the first word in a document, it would not have needed its extra capitalization recording byte. If these latter assumptions were all correct, the result would be an overall 60% compression rate. In our implementation, the compression ratio falls somewhere between 50% and 60%. The average in tests to date is about 55% (i.e. better than a factor of two).

3. APPLICATIONS

There are many applications for a compression method such as this one. Its speed is much faster than that of more general binary compaction algorithms. Since it is word oriented it can be used as a filter,

```
This, Mr. Smith, is a simply amazing sample sentence.
##@ - --- ------ *  - ##@    ##@     ##     ##      -
```

FIGURE 1

uncompacting one word at a time and then passing it back to a calling routine which need not know that the word was ever compressed. The function

```
funcompact(source,"%s", stringvar)
```

can execute almost as quickly as

```
fscanf(source,"%s",stringvar).
```

3.1. Serial communications

The greatest realization of the advantages of this compression algorithm is achieved when it is used in conjunction with communications across a modem or other serial adapter. Using an intelligent terminal, one can transmit documents in compressed form, and have the recipient decompress them. This has the affect of doubling the baud rate. We can decompress words much more rapidly then we can transmit words at current baud rates. Text oriented services can apply this method to great advantage, particularly information retrieval services which deal exclusively in text transmission. Doubling the effective transmission rate is extremely noticeable over phone lines which currently have a maximum transfer rate of 2400 baud.

3.2. Online documentation

Using this method of compression on documentation stored on-line has the advantage of saving disk space without sacrificing speed of access. There is a cost to this, however, since more processor time is required to decompress a file than to merely list it. However, this is not the case with systems which store documents in unformatted form, and then format them when they are retrieved. Compared to most formatting schemes, decompression is faster, and tokenization provides greater compression. Expensive memory can be saved by storing unformatted documentation offline and only keeping compressed, formatted documentation online.

3.3. Text searching

Built into tokenization are many features which greatly facilitate searching text documents for a given word. Since words are compacted into distinct segments: word, capitalization attributes, prefixes and suffixes; much of the work involved in searching is already taken care of. By first encoding a query word, compressed text of interest can be searched numerically (for an identical index) rather than by string comparison. Capitalization and prefix and suffix stripping has already been done. Furthermore, the number of characters to be searched is greatly reduced.

4. DATA

One feature of this compression scheme is that it can easily be "tuned" to different environments (this is discussed in more detail later). Many of the rules, and the dictionaries themselves, are external to the compression and decompression routines. That is, the code itself is not dependent on a specific dictionary or rules, but merely on a specific data format. Consequently, the effective implementation of this algorithm is highly dependent on the form of the data.

There are five static data lists used by both the compression and decompression programs which contain the data which must be shared between the routines. Since in tokenization it is pointers or indices which are passed, it is essential that the same lists be used when compressing and decompressing. Hence, in order to change dictionaries, either every document must be decompressed using the original dictionary and then re-compressed using the new one, or a copy of the old decompression program must be maintained.

There are three basic categories of lists: the word lists, the prefix and suffix lists, and the character attribute list. These are detailed below.

4.1. Words

As mentioned before, two separate word lists are maintained. One is a large dictionary of 8K (8192) words and the other contains the 32 most common words. Whenever an attempt is made to look up a word, the short list is searched first, and if the word is not found the longer list is searched. The shorter list is maintained for two reasons: it is quicker to search (a word is more likely to be found in the short list than in the long list) and it provides a clean method of encoding words into only one byte. To not transfer words in only one byte would be a tremendous waste. The most common word in the dictionary occurs almost 7,000 times more often in documents than the least common and 100 times more often than the thirty-third most common (see [1]).

The short list is maintained as a simple array of null terminated character strings in alphabetical order. Words are looked up via a binary search.

The long list is slightly more complicated. It is stored in alphabetical order by first letter, then sorted by length within each group having the same first letter. Finally, each length is put into alphabetical order. The following words are in the correct order:

```
ape
apple
apply
applied
addition
boo
baby
bake
badger
bailiff
```

In order to optimally search this list, another list is maintained which contains pointers (offsets) into this list. If one is attempting to look up a given word of known length, this list points to where the search should begin. This header list is referenced by an index corresponding to the length of the word and its first character. The actual formula is

```
index=(first_character-'a')*(MAXWORDLENGTH-2)+length-3
```

where first_character is the first character of the word being looked up, MAXWORDLENGTH is a constant equal to the maximum allowed length of a word in the dictionary (words longer than MAXWORDLENGTH can not be in the dictionary so are not searched for), and length is the length of the word being looked up. If MAXWORDLENGTH equaled 12, the index of the header pointing to the word "cabbage" would be

```
('c'-'a')*(12-2)+(7-3) = 2*10+4 = 24.
```

This header list is a structure of the type

```
  struct header_type {
    int offset,  /* byte offset into the list for words of this type */
      total,     /* number of words with this length and first char. */
      index;     /* of the first word of this length                 */
}
```

Offset refers to an actual character offset, so in the earlier example "apple" would have an offset of 3 and an index of 1, while "ape" would have an offset and index both of 0. The structure of the header offers one additional advantage. It is not necessary to carry the first character and a terminating character for each word. Therefore, the entire list of 8K words is maintained as one long character array, with no first characters or terminating nulls. Again referring to our earlier example, this list would look like

```
char long_list="pepplepplypplied..." ;
```

Since for a given length and first character, one knows the number of elements (courtesy of the header structure), a binary search can still be employed. (Note that the offset of "apple" is now 2.)

The memory savings involved in this are tremendous. By eliminating two characters per word (first character and separator), 16K bytes are saved; by not implementing it as an array of pointers another 8K times the pointer length is saved (possibly 32K bytes). Using this scheme, our header table and word list together occupy just 53K bytes. This is a savings of 12K bytes when compared to conventional access methods.

4.2. Affixes

Two additional lists (each with two parts) are maintained for prefixes and suffixes. Each list is divided into two parts. This first part contains the 16 most common prefixes or suffixes and the second the additional affixes. Each part of each list is independently put in alphabetical order. Suffixes are ordered by the last letter of the suffix. Those prefixes and suffixes in the first part of each list are those whose index can fit into four bits and thus be carried along with other information. The others must be tokenized with a dedicated byte. Each part of each list must be searched separately since they are ordered independently.

All searches of affix lists must be unary in order to avoid conflicts between prefixes such as "un" and "under" or suffixes like "es" and just plain "s". Both lists are arrays of character strings, with the length of the affix carried in binary before the actual affix:

```
char *prefix_list[] = {
    "\004anti",
    "\003dis",
        .
        .
        .
    "\005ultra"
} ;
```

Additionally, suffixes carry (in the same byte as the length) information on which root-modification rules apply to it. Prefixes and suffixes, like words themselves, are coded on compression as indices into the appropriate list.

4.3. Character attributes

This final list serves a different purpose than the others in that it is not used to determine token indices for compression. Instead, this list contains information on each ASCII character and the assumptions made about it (i.e., space before or no space before). It is implemented as an array, and the attributes of a given character can be referenced as ch_attrib[ascii_value]. Each bit of the value represented by ch_attrib[ascii_character] indicates the presence or absence of a given property. These properties are

0) space expected after,
1) space expected before,
2) end of sentence expected,
3) is punctuation,
4) is numeric,
5) is alphabetic,
6) is white space,
7) is binary.

All one need do to determine if a given character contains a certain attribute is perform a bitwise "AND" between the value of the array subscripted by that character and the number resulting from setting only the bit of interest in the attribute. For example, if one wanted to determine if the period mark (.) was punctuation, and bit 3 indicated punctuation, then

```
ch_attrib['.'] & 00001000b
```

would be non-zero since a period is, in fact, punctuation. If a period were not punctuation, the value would be zero.

Again it is important that this array be identical in both the compression and decompression routines, lest they make different assumptions about a character.

5. Compression

5.1. Overview

Following is a brief description of how the compression part of the algorithm works. The program has two distinct stages: one in which a word is parsed and another in which it is processed. The type of processing done on a word depends on the type of the characters making up the word.

5.2. Input

Text is parsed by the program into units termed "words". A word need not be an actual English word, but merely a contiguous grouping of characters of the same classification (alphabetic, numeric, white space or punctuation).

This algorithm is word oriented in the sense that each word is independently parsed, processed and output before another is accepted. The following are the character classifications for reading in words:

1) alphabetic characters,
2) numeric characters,
3) punctuation, and
4) white space.

To facilitate the parsing of words, the character attribute array is used as shown in Figure 2.

5.3. Processing

Once a distinct word is identified, the program attempts to compress it. The nature of the compression depends on the type of the word. Each classification and each word are handled independently. Figure 2 shows how the initial identification of a word and the processing of it are integrated. The specific methods of handling each type of word are outlined below. There is some data which must be shared between the various "word" handling routines. This is

```
char    *wbegin, *wend              /* bounds of current word         */
        *from,                      /* source of text to compress     */
        classify(),                 /* returns 1 bit set to type of word*/
        its_type;                   /* 1 bit set to type of word      */

do {                                /* begin compressing              */
    wbegin = wend = from;                       /* reset boundaries */
    its_type = classify(*wend);                 /* get the type     */
    while(token_type[*++wend] & its_type);      /* find word's end  */
    from = wend;                    /* set input to start of next "word"*/

    switch (its_type) {
        case ALPHABETIC: ...
        case NUMBER: ...
        case PUNCTUATION: ...
        case WHITESPACE: ...
    }
} while(*from !='\n');              /* compress until newline */
```

FIGURE 2

accomplished via several global binary variables. These variables are

last_skipped -- set if a space was not compressed because it was assumed,
next_cap -- set if the next word is expected to be capitalized,
space_next -- set if a space is expected next (e.g. following a word), and
last_not_in -- set if the last word could not be tokenized.

With the exception of these variables (actually bit fields within a single byte), each routine can function independent of the rest of the program.

5.3.1. Alphabetic Strings

Alphabetic strings are assumed to be English words, and every possible attempt is made to tokenize them. In order to maximize the chances of a word being tokenized the following procedure is employed, in the given order, until a match is found. A search is performed after each step if the step applies to the word:

1. The word is looked for in the short dictionary,
2. Case is made all lower case,
3. Any prefix is stripped,
4. Any suffix is stripped and any prefix is put back on the word,

THE FOLLOWING STEPS ARE PERFORMED ONLY IF THERE WAS A SUFFIX:

5. If there was a prefix it is removed from the word again,
6. Any prefix is restored and the root is modified if possible, and
7. If the root was modified the word is stripped of any prefix.

If a word has still not been found, it is not compressed but placed literally in the compressed document. Note that for some words, not every step need be taken. Steps (4) through (7) apply only if the word has a suffix. Steps (3), (5), and (7) apply only if the word had a prefix. Hence a word with no prefix or suffix only encounters steps one and two. An example of the type of word for which multiple searches would be performed is "Unhappily". It contains a capital "U", a prefix "un", a suffix "ly" and the "y" in "happy" was changed to "i". The following suffix transformations are recognized:

1) 'y' to 'i' before 'es', 'ness' and 'ly',
2) doubled consonant before 'ing', 'ed' and 'er',
3) 'y' to 'i' in cases other than (1), and finally,
4) final 'e' being dropped.

These rules are checked in the listed order. Note that in some cases these rules are applied without checking to see that the rules of English have been correctly followed. (i.e., if a word ends in 'i' and makes it to step (3), it will automatically be changed to 'y'). This is because the objective of this algorithm is not to be a spelling checker, but to compress words even if they are not spelled correctly.

5.3.2. Numbers

One special case of numbers is currently recognized. Numbers between 1800 and 2055 are assumed to be years and are compressed as a year flag followed by an offset (year-1800). These numbers are treated as if they were tokenized words, while other numbers are handled the same as untokenized words (i.e., words not found in the dictionary).

5.3.3. White space

There are two types of white space: actual ASCII spaces; and tabs, carriage returns, form feeds and the like. Other than actual spaces, white space can not be eliminated. However, strings containing four or more repeated, consecutive, non-printing characters are compressed into a tag byte, a count byte, and the actual character. Three or fewer repeated characters are passed literally.

Stand alone spaces also can be dealt with. If a space separates two words (at least one of which must be tokenizable) it can be eliminated. If a space occurs before a punctuation mark defined as having a space before it, it need not be carried. Additionally, if the space occurs after a mark defined as having a space after it, the space also need not be passed along.

5.3.4. Punctuation

Single punctuation marks must always be passed literally. Four or more repeated, consecutive punctuation marks are handled in the same manner as repeated white space. They are compressed into a header byte, and a count byte followed by the punctuation mark. Punctuation does affect the compression of the other types of words. Both the first and last character(s) (they can be the same) of each punctuation string are analyzed, and flags set accordingly. For example, if a word ends in a period, the end of sentence flag is set. Or, if a space was expected, and instead an open parenthesis (which assumes a preceding space) was found, the punctuation handler outputs the rule override character.

6. DECOMPRESSION

With few exceptions, decompression is much more straightforward then compression. Because indices are passed, this part of the program merely has to recognize and put the affixes in the right place, and know the assumptions that went into the compression.

The most time consuming aspect of decompression is to locate a word in the large dictionary given its index (recall that this dictionary is maintained as one 50K byte character string without first characters or word terminators). Figure 3 shows how this is accomplished. The length and first character of a word can be derived from the index of the member of the header table which refers to that word by the following formulas:

```
/*
 * The following routine finds the index of the element in the header
 * table which refers to the word whose position in the dictionary is
 * given by 'dp'
 */
int dp,          /* the position of the word of unknown header     */
    first,       /* index of 1st word pointed to by current header */
   guess,        /* guess for current header subscript             */
    high,        /* upper bound for binary search                  */
    low;         /* lower bound for binary search                  */
extern struct header_type table[];      /* the header table itself*/

/*
 *  A binary search is used to determine the header table subscript
 *  corresponding to the given index:
 */

low=0;
high = TABLESIZE;
while (low <= high) {           /* find the header for the index */
    guess=(high+low)/2;
    first = table[guess].i_index;
    if (dp < first) high = guess-1;         /* guess is to high */
    else if (dp > first) low = guess+1;     /* guess is too low */
    else {                                  /* GUESS IS CORRECT */

        /*
            The following loop passes up potentially empty header
            records. A header record gives the index for the first
            word in a list of words with a given length and first
            character but there is no guarantee that there are
            actually any words of that length and first character.
        */

        while (table[guess].total==NULL) guess++;
        high = guess;
        break ;
    }
}
/* 'high' now contains the subscript of
 * the header table member which refers to 'dp'
 */
```

FIGURE 3

```
length = (index % MAXWORDLENGTH) + 1

first_character = index/MAXWORDLENGTH + 'a'.
```

7. ADAPTATIONS

Another feature of this compression scheme is that it is easily adaptable to many different types of document compression. Without changing the code, new data (words, affixes, and punctuation attributes) could be integrated into a system by merely re-linking the programs with the new data. Hence, several versions of the compression program could be maintained, each for a different application. It might not be appropriate to expect the same dictionary to be used to compact technical online documentation and to compact a philosophy thesis. Because different programming languages use punctuation in different ways, savings can be maximized by maintaining different versions of the compression program with different rules for each language.

- APPENDIX A -

COMPRESSED DATA FORMAT

Below is the actual format used to tokenize words. Bit zero is always the least significant bit, and seven is the most significant bit.

FIRST BYTE OF TOKENIZED WORD

bit	description
7	always 1 to indicate that the word is tokenized
6	1 if tokenized to two bytes, 0 if to one
5	1 if there are attribute bytes, 0 if there are not
4	} Bits 0 through 4 contain the actual index of the word.
3	} If it is a one byte tokenization, these bits are the
2	} complete index (in binary), otherwise they are the low
1	} order five bits of the index into the 8K dictionary.
0	}

SECOND BYTE OF TOKENIZED WORD:

If the word was tokenized as two bytes, this second byte contains the high order eight bits of the index. The index can be reconstructed by the following formula:
index = first_byte + second_byte<<5
If the word was tokenized into one byte, this byte is omitted.

FIRST ATTRIBUTE BYTE (SECOND OR THIRD BYTE)

bit	description
7	Bits 6 and 7 signal capitalization. The keys are: 00=use default
6	01=first letter cap, 10=whole word cap, 11=all lower case.
5	1 if there are more attribute bytes, 0 if this is it
4	1 if a prefix is encoded in the following bits, 0 if a suffix is
3	
2	Bits 0 through 3 contain the index of either a prefix or suffix,
1	if there was any. 0000 is an illegal suffix so that if all of bits
0	1 through 5 are zero, it indicates that no affix is encoded here.

REMAINING ATTRIBUTE BYTES

The remaining attribute bytes can take on either of the following formats. The purpose of these different formats is mainly one of convenience: Prefixes and suffixes placed into the shorter of the two formats can be more easily concatenated into a first attribute byte along with the capitalization attribute.

SHORT FORM

The short form is identical to the encoding of the first attribute byte, with the exception being that the two high bits (previously used for capitalization), are always set to 0.

LONG FORM

bit	description
7	0 if a prefix follows, 1 if a suffix
6	always 1 for a prefix, normally 0 for a suffix, unless the root of the word was modified to accommodate the suffix, in which case this is 1.
5	1 if there is another attribute byte, 0 if there is not
4	}
3	} Bits 0 through 4 contain the binary encoded prefix
2	} or suffix (00000 is legal for both prefixes and suffixes).
1	}
0	}

- APPENDIX B -

STATISTICAL RESULTS

The following statistics were gathered by a 'C' program running under UNIX on a MicroVAX I. The short dictionary is composed of the 32 most commonly occuring words in the English language (see [1]). The long dictionary contains the next 8K most common words. The documents are all newspaper articles. All sizes given are in bytes, and the cpu times are in seconds.

size		percentage	cpu time required	
uncompressed	compressed	compression	compression	decompression
15311	6619	56.8	10.1	3.9
8675	3749	56.8	6.0	2.6
6214	2687	56.8	4.4	1.6
5086	2087	59.0	3.3	1.3
1758	755	57.1	1.1	0.5

The above five files taken together contain 5592 words (since the dictionaries were not prepared for these documents, not all of these words were contained in them). They require a total of 24.9 seconds to compress and 9.9 seconds to decompress. Hence, 225 words/second can be compressed and 565 words/second decompressed.

Text Compression Using Word Tokenization

The results in the following example were obtained using this paper itself. These results exemplify the advantages of storing compressed rather than unformatted manuals online.

size			time	
formatted uncompressed	unformatted	formatted compressed	to format	to decompress
39027	33719	18032	229.7	9.5

REFERENCES

[1] Francis, W. N., Kucera, H., Computational Analysis of Present-Day American English, Brown University Press, 1970.

[2] Kelly, E., Stone, P., Computer Recognition of English Word Senses, North-Holland Publishing Company, 1975.

[3] Warriner, John E., Treanor, John H., Laws, Sheila Y., English Grammar and Composition, Harcourt, Brace & World, Inc., 1969.